LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-656718

MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach

J. Protze, T. Hilbrich, M. Schulz, B. de Supinski, W. Nagel, M. Mueller

July 8, 2014

Fifth International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2014)
Minneapolis, MN, United States
September 9, 2014 through September 14, 2014

Disclaimer

MPI Runtime Error Detection with MUST: A Scalable and Crash-Safe Approach

Joachim Protze*, Tobias Hilbrich†, Martin Schulz‡, Bronis R. de Supinski‡,
Wolfgang E. Nagel† and Matthias S. Müller*
*RWTH Aachen University
D-52056 Aachen, Germany
Email: {protze, mueller}@itc.rwth-aachen.de
†Technische Universität Dresden
D-01062 Dresden, Germany
Email: {tobias.hilbrich, wolfgang.nagel}@tu-dresden.de
‡Lawrence Livermore National Laboratory
Livermore, CA 94550
Email: {bronis, schulzm}@llnl.gov

***Abstract*—The Message Passing Interface (MPI) is a widely used paradigm for distributed memory programming. Implementations of this interface are designed for good performance rather than on usability extensions that enforce their correct use. Runtime MPI usage error detection tools aid application developers in the correct use of this interface. Since usage errors can cause failures that lead to an application crash, it is crucial that runtime error detection tools employ techniques that allow them to finish all of their correctness checks. This includes situations in which the application is interrupted by the MPI library, due to an incorrect function call; and operating system signals after fatal errors like division by zero or faulty memory accesses. We present an approach that uses an alternative tool communication means along with signal and error handling capabilities. A study of the assumptions that enable this approach details its applicability for different use cases and compares it to less efficient schemes that rely on synchronous processing and/or communication. Additionally, we enable bandwidth efficient communication with a scalable propagation technique that raises the awareness of an application crash within the tool. An application study with the SPEC MPI2007 benchmark suite demonstrates the applicability of our approach for up to 2,048 processes. Overhead measurements underline that our application crash handling increases the runtime of our runtime error detection tool by only 4% in average.**

I. Introduction

The Message Passing Interface (MPI) [1] provides a portable and widely used library specification for parallel programming on distributed memory systems in High Performance Computing (HPC). The interface specification of the widely implemented version 2.2 consists of 700 pages already. This specification both defines the behavior of the various functions of the interface as well as constraints and preconditions for their intended use. The behavior of an MPI implementation for cases where an application breaks any of these constraints is undefined and represents a challenge for the development of correct and portable applications. Tools that reveal usage errors of MPI (*defects*) support efficient development workflows. We develop MUST [2] as one such tool that operates during the runtime of the application. The tool detects wide ranges of MPI usage errors such as MPI datatype matching violations, incorrect communication buffer use, and deadlocks.

Figure 1 illustrates an MPI usage error that calls the collective operation `MPI_Reduce` with four application processes (ranks 0–3). The illustration leaves out any initialization and finalization operations as well as function arguments that are irrelevant for this error situation. In the scenario, all collectives specify the same *root* rank, that is, all processes consistently agree that rank 0 will retrieve the final result of the reduction. However, rank 0 specifies a different datatype than the other ranks, which violates the MPI type matching rules. In practice—especially for larger communication payloads—such defects can cause an application crash or trigger error handling code within an MPI implementation or another software component. We use the term *failure* to refer to these error situations and *crash* to refer to the abortion of execution. Thus, a runtime correctness tool for MPI must detect an MPI usage error in the presence of failures. A simple solution to this challenge is a correctness analysis that occurs in the critical path of the application, i.e., that detects an MPI usage error before it can cause a failure. Such an analysis comes with a severe performance and scalability penality for error situations that require information from multiple processes. The situation in Figure 1 is one such example where a correctness tool needs to combine information from rank 0 and at least one further rank to reveal the defect. To overcome this situation and combine failure tolerance with an efficient and scalable correctness analysis we:

- Highlight and compare solutions to handle failures,
- Select and implement a solution that uses multiple communication mediums while it retains a high bandwidth,
- Provide methods to both enable buffering (high bandwith) communication in a TBON (tree-based overlay network) tool and to shut down a tool after a failure, and
- Study the effectiveness and the performance impact of our approach in a prototype implementation with the correctness tool MUST.

Particularly, our solution to use multiple communication mediums is not only applicable to TBON-based tools, but

Process 0	Processes 1–3
`MPI_Reduce` (type:`MPI_INT`, root:0)	`MPI_Reduce` (type:`MPI_DOUBLE`, root:0)

Figure 1. MPI usage error that can cause an application failure.

rather to any runtime tool. The methods to handle buffering and shutdown, then detail extensions to this approach that are applicable to TBON-based tools. We detail the problem of failures in runtime (and TBON) tools in Section II. Afterwards, we compare different failure handling solutions in Section III and highlight the approach that we selected for implementation in Section IV. Section V then details the TBON methods that enable the use of buffering communication (for bandwith efficiency) in the presence of failures along with a method to shut down the tool. A study with a complex and widely used benchmark for MPI then analyzes the performance impact and the applicability of our approach in Section VI. Finally, Section VII compares our approach and methods to related work.

II. TBON Tools for MPI and Failures

The runtime tool MUST targets low overhead and a high scalability along with deadlock detection capabilities. As a result, it uses an offloading concept to avoid an analysis in the critical path of the application:

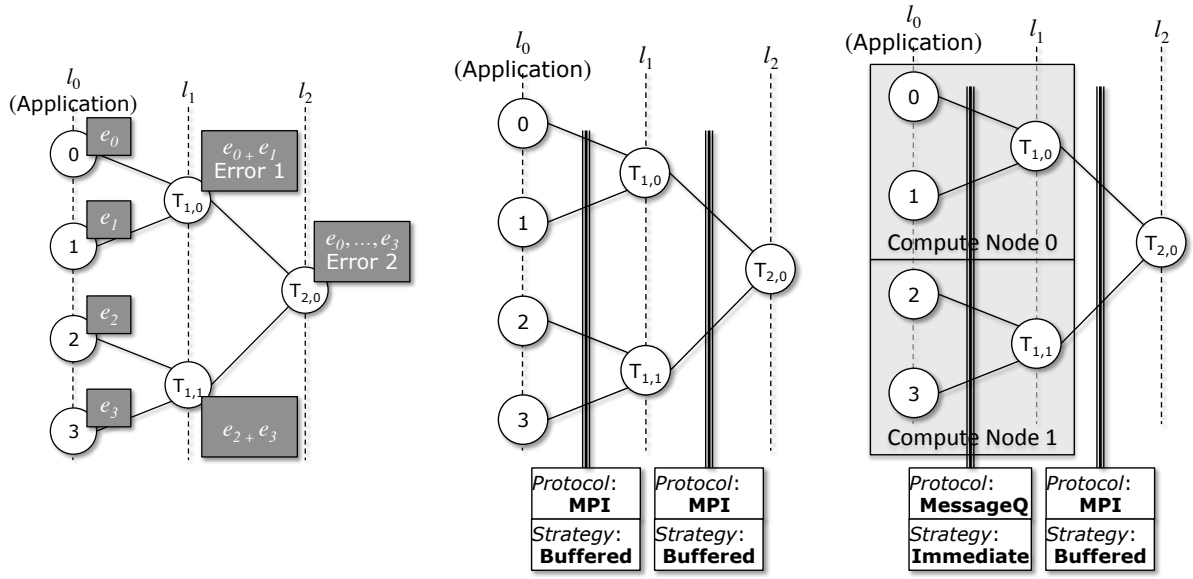
- On the application processes, MUST instruments MPI function calls and packages information on them into events, and
- It uses additional processing elements to analyze these events in a distributed and scalable manner.

Since correctness analyses such as deadlock detection, and tool activities such as logging MPI errors require a global view, MUST uses the TBON concept [3]: Events travel through a tree network whose root can gather global information and whose intermediate layers provide hierarchical event processing capabilities. Figure 2(a) illustrates a TBON for the four application processes from the example defect in Figure 1. The illustration represents the ranks as nodes with labels 0–3, the root of the TBON as a node with the label $T_{2,0}$ (first node in the third layer), and two nodes of an intermediate layer with labels $T_{1,0}$ and $T_{1,1}$. The arcs between the nodes highlight the communication links. In the case of MUST, the usual event flow is from the application processes towards the root of the TBON, e.g., rank 0 would forward its events to $T_{1,0}$. The Figure highlights events e_0–e_3 that shall represent information for the `MPI_Reduce` operations from Figure 1. Thus, node $T_{1,0}$ would perceive information on events e_0 and e_1, i.e., on the reduction operations of the first two ranks. Node $T_{1,1}$ perceives information for events e_2 and e_3, while the root perceives information on all events. This distribution of the events highlights that both nodes $T_{1,0}$ and $T_{2,0}$ can detect the MPI usage error, but that it can not be detected directly on the application processes. This situation highlights the overall setting in which we consider failures on application processes. However, this offloading setting is not unique to MUST but reflects the operation mode of other MPI correctness tools (e.g., ISP [4]). Also, debugging tools (e.g., STAT [5]) and performance analysis approaches (e.g., Periscope [6]) process events in TBONs and could benefit from application failure handling schemes.

MUST analyzes all MPI function invocations for their correctness. Its communication within the TBON requires high bandwidth and/or low latency (depending on the use case) since communication operations can be frequent and their analysis expensive. Additionally, the maintenance and development of portable TBON services is challenging. As a consequence, MUST utilizes the tool infrastructure GTI [7] that provides TBON services. GTI provides a plugin mechanism that enables support for multiple communication mediums. We use the term communication *protocol* to refer to a communication medium plugin. GTI's primary communication protocol for MPI tools is a communicator virtualization technique [8] separating the initial MPI_COMM_WORLD communicator to an application world communicator and tool processes; this approach utilizes MPI communication operations. This allows MUST to use the highly optimized communication capabilities of most HPC systems, without the use of restricted bandwidth I/O nodes [9]. In addition, GTI provides a flexible choice in communication timing. A plugin mechanism allows flexible choices between so called communication *strategies*. A strategy can pass new events immediately to a communication protocol for low latency communication or it can aggregate multiple events into larger continuous buffers for bandwidth efficient communication. Figure 2(b) illustrates common communication choices for MUST. GTI allows a tool to use specific protocol-strategy pairs between individual hierarchy layers where MUST commonly uses an MPI-based communication protocol and a bandwidth efficient (*buffered*) strategy between all hierarchy layers.

Figure 3 illustrates the issues that arise for such a communication setting if failures occur on an application process. Assume that the error situation in Figure 1 causes a failure on rank 0, due to the use of an incorrect MPI datatype. The figure sketches the processing within rank 0 and the TBON node $T_{1,0}$ as a sequence diagram. MUST would intercept the `MPI_Reduce` invocation on rank 0, forward an event to $T_{1,0}$, and then invoke the actual MPI operation (`PMPI_Reduce`). The communication of the event is asynchronous and the default communication selection in MUST returns the control flow to rank 0 before the event arrives on $T_{1,0}$. The illustrated application failure that occurs when rank 0 issues the actual MPI operation has the following consequences in MUST's default setup:

- Since the failure occurs within MPI, the tool can't rely on the asynchronous event communication to finish, as it uses the MPI implementation that is now in an undefined state,
- The communication may not even be initiated yet due to the buffering communication strategy, and
- The failure on rank 0 can trigger timeouts in the MPI implementation that cause a global abort of all MPI processes, i.e., also of the TBON nodes (due to GTI's MPI communicator virtualization all application processes and all TBON nodes represent one application).



(a) Event sources and event processing. (b) Default communication choices. (c) Proposed multi protocol solution.

Figure 2. Illustration of a TBON-based runtime correctness tool for the error situation of Figure 1 (e_0–e_3 represent the `MPI_Reduce` operations of processes 0–3).

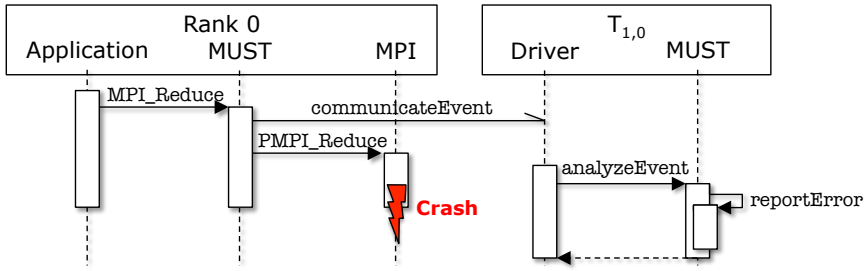


Figure 3. Application failure (within MPI) and asynchronous processing in a TBON for the runtime correctness use case.

A crash handling scheme for runtime MPI tools such as MUST (with or without a TBON) must overcome these deficits to allow a tool to analyze all events that occured prior to a failure. For the situation in Figure 3, a mechanism should ensure that $T_{1,0}$ can receive event e_0, analyze it and any other event (e_1) that the tool needs to report the MPI usage error.

III. Crash Handling Strategies

An application crash handling scheme for an offloading-based runtime tool must satisfy the following requirements:

A) (*Information*) Avoid the loss of event information,
B) (*Continuation*) Ensure that all tool analysis can continue, and
C) (*Communication*) Existing event information can still be communicated within the tool.

A common approach to meet these goals is an event analysis in the critical path. This approach ensures that analysis finishes before a crash occurs. Such an approach can cause high tool overheads and limit scalability at the same time. More efficient approaches allow some degree of asynchronous communication or processing by using specific failure handling mechanism.

A. Synchronous Communication

The purely synchronous technique handles an application crash by communicating and analyzing all events in the critical path: As to avoid loss of information, a tool immediately processes events when they occur. Thus, when a tool perceives a new event, it returns the control flow only after it finished all event processing. When an application crash occurs, the tool already analyzed all exisiting events and thus trivially meets Requirements A-C.

The event handling in Figure 3 underlines the necessary steps to enforce the purely synchronous handling. The event communication (`communicateEvent`) and the event processing on $T_{1,0}$ must complete before rank 0 issues `PMPI_Reduce`. This requires an acknowledgement mechanism that provides rank 0 with information on when $T_{1,0}$—and potentially further TBON nodes—processed the new event. In the case of MUST, any MPI communication call can cause a crash. Thus, before an application process could issue an MPI operation, the tool needs to finish its correctness analysis for the operation in question. Correctness analyses like deadlock analysis require complex communications within the TBON and can spread over multiple hierarchy levels. An acknowl-

edgement mechanism that ensures that event processing occurs in the critical path would introduce long processing delays for MUST, thus causing severe tool overheads. Additionally, providing feedback when the analysis of an event is finished within the TBON in a manner that does not introduce deadlock is challenging.

As a consequence, existing tools such as Marmot [10] employ this strategy, but it causes high overheads and complex interactions for a TBON-based tool.

The synchronous communication technique weakens the restrictions of the previous scheme in that it only communicates in the critical path of the application, but processes events asynchronously. Additionally, only the application layer must use synchronous communication, while all remaining layers can utilize asynchronous communication. In the example of Figure 3, this requires that rank 0 only continues its execution after `communicateEvent` transfered all event information to $T_{1,0}$. In contrast to the first strategy, rank 0 does not need to wait until $T_{1,0}$ and any other TBON nodes analyzed the event. This decreases the amount of necessary synchronization. Since application processes transfer event data first, this scheme immediately satisfies Requirement A (*information*). However, event processing may still be ongoing when a crash occurs. Thus, mechanisms or restrictions in the tool must ensure Requirements B and C (*continuation* and *communication*). Thus, we employ the following assumptions for our tool:

I) A crash only occurs on the application processes;
II) The application processes only create events, but never receive any events; and
III) A crashed application process does not:
   a) Cause crashes of higher level TBON nodes, nor does it
   b) Impact communication on other TBON layers.

The first two assumptions are necessary since a crashed application process will not be able to receive and process any outstanding events. Thus, we can meet Requirements B and C only if application processes do not need to receive events. Additionally, we restrict crashes to application processes to avoid the need for crash recovery of tool nodes in the TBON. These first two assumptions apply to MPI correctness tools such as MUST and do not limit its functionality. Tools that offload events from the application processes commonly try to avoid communication towards application processes, due to the resulting control flow impact, and if so usually employ extra threads on the application processes.

Assumption I can become unsatisfactory in a failure prone environment that regularly causes failures on non-application TBON nodes. Existing concepts for TBON tools [11] provide an approach to extend failure handling capabilities to these TBON nodes as well.

Assumption IIIa ensures that the crash handling technique satisfies Requirement B. This assumption usually requires specific mechanisms—details on the MUST implementation follow—since most MPI implementations recognize situations in which an MPI process crashed. If so, they usually abort all other processes, thus, causing an indirect abort (crash) for other TBON nodes. Assumption IIIb requires that if an application process crashes, it does not impact communication between TBON nodes of higher hierarchy layers, thus, guaranteeing Requirement C. In the example of Figure 2(a), if Assumption IIIb holds, a crash on rank 0 should not impact communication between $T_{1,0}$ and $T_{2,0}$. As long as a tool purely employs MPI communication, this assumption may not hold for all types of failures nor for all MPI implementations.

This second strategy decreases the performance impact of the crash handling scheme, but guaranteeing Assumption IIIb is non-trivial as long as a tool purely relies on MPI communication. Additionally, the communication in the critical path of the application can still cause noticable overheads.

B. Asynchronous Communication

The completely asynchronous technique extends the previous technique to both remove the need for synchronous communication and to minimize the communication system impact of a crashed application process (Assumption IIIb). Since an MPI communication related crash on an application process forbids the asynchronous use of MPI to transfer events to other TBON nodes, we must rely on a second (alternative) communication medium. Figure 2(c) highlights this for GTI where we use a shared memory communication protocol between the application layer and the first TBON layer. All remaining layers can still employ MPI communication. In the case of the crash situation of Figure 3 we would use asynchronous (but non-MPI) communication to implement `communicateEvent` and issue `PMPI_Reduce` while the communication and the event analysis is still outstanding. This approach reduces the performance impact of the crash handling scheme drastically, but relies on an additional assumption:

IV) A crashed process can complete asynchronous communication on the alternative communication medium.

Besides the performance improvement from the asynchronous communication, the alternative medium decouples the application layer from the remaining tool layers, i.e., no application process communicates with a non-application process via MPI. This property impacts the feasibility of Assumption IIIb, since no outstanding MPI communication will exist between an application process and any non-application node.

An application failure has additional impact if a tool uses buffering communication as for MUST. In that situation, once a failure occurs, the tool must forward all events and avoid future buffering. Section V details how we notify all TBON nodes of a failure at an application process to achieve this.

C. Alternate Communication Means

There are two basic approaches to integrate an alternate communication system:

- Swap the communication means after a failure occured, or
- use the alternate communication system from the beginning

Swapping the communication means after a failure occured has the advantage that we benefit from highly optimized MPI communication while the program works well and only employ the fallback if a failure occurs. However this approach

comes with drawbacks: During a failure we must propagate the information that a communication swap is necessary to other processes. For that we must not use the default communication means. As a result, we have to initialize and poll on the alternative communication at all times (before a failure occurs). More critical are cases where a process tries to use a the default communication means while it is in an invalid state, due to a failure and before a notification arrives. Such a communication attempt could lead to a failure on non-application processes.

D. Implementation in MUST

We implement the completely asynchronous crash handling technique in MUST and rely on a shared memory communication as the alternative communication medium, which we describe in Section IV. Our use case satisfies Assumption II and we limit our crash handling to application processes (Assumption I), where future extensions could add failure recovery for intermediate TBON nodes. In order to satisfy Assumptions IIIa and IV, we must catch crash situations. This serves to flush all messages that are in transit on the alternative communication protocol and to avoid that the MPI implementation aborts the whole application. Note that we can not completely guarantee Assumptions IV, if a malicious memory access yields an inconsistent state for our alternative communication means or a component that it uses, then an attempt to communicate with this means could fail. However, a small subset of failures should yield such a situation.

We catch error situations the following two kinds of failures: Errors reported by the MPI library and errors detected by the operating system, which result in a signal, e.g., `SIGSEGV` for a malicious memory access. To catch MPI errors we register an error handler with the MPI library. For signals from the operating system we register a signal handler. Depending on the operating system, the programming paradigm of the application, and the runtime system, further types of signals or error handlers can indicate a failure. The above selection of handlers allows us to catch crash situations in practice on various Linux clusters. We propose a try-catch-throw mechanism based on *setjmp* and *longjmp* functions of POSIX to guard call of MPI functions. This allows us to distinguish a failure that results from an incorrect call to the MPI library from errors within the application. The output of MUST includes function call stacks for detected MPI usage errors. The try-catch-throw mechanism restores a valid function call stack, so we may include the failure information and the stack trace to the report.

Finally, as stated above, we rely on the MPI implementation to guarantee Assumption IIIb. The alternative communication protocol ensures that no outstanding communications exist between application processes and other non-application nodes of the TBON. In practice we find that the MPI implementations that we investigated—MVAPICH, Open MPI, and BullXMPI—satisfy our assumption.

IV. Shared memory communication protocol

As introduced in Section II, our tool infrastructure GTI uses MPI as its default communication means. The completely asynchronous crash handling technique requires that we do not rely on MPI for communication past a failure.

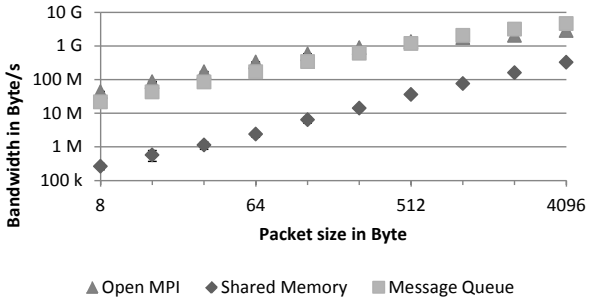


Figure 4. Point-to-point bandwidth comparison of message queues, shared memory blocks and Open MPI

Many of todays computing clusters are built with SMP nodes that contain a number of computing cores with shared memory. We exploit this architecture by placing each first level tool proces and its connected application processes onto one compute node, such that these processes have available shared memory. We then provide a shared memory implementation of the GTI protocol interface based on System V IPC (inter process communication). This protocol is used for communication between the application layer l_0 and the first non-application layer l_1, as illustrated in Figure 2(c).

System V IPC provides three basic paradigms:

- Message Queues
- Semaphores
- Shared Memory

Message queues are typically limited by the operating system in space to a few kBytes and in count to a few hundred or thousand messages; there is no explicit message count limit, but an implicit limit is given by the space limit. Message queues provide a convenient way to implement consistent send and receive transfers with atomic read and write operations. We use two message queues for each communication channel, one for each direction; a channel is equal to an egde in the TBON.

Since message queues are limited in space we use *shared memory* chunks to transmit larger messages. For *shared memory* chunks there is a limit in count given by the operating system—typically a few thousand—while space is limited by system memory only. For large messages we allocate a shared memory chunk first and copy the payload into the chunk. Afterwards, we transfer a message (via the message queue) to pass the reference to the receiver side. Thus, we need no further explicit locking mechanism (*semaphores*) while we preserve message order at the same time.

Figure 4 compares throughputs for direct communication via messages queues (*Message Queue* in the figure) and for explicit shared memory allocation with passing a reference through the message queues (*Shared Memory* in the figure). The former communication type provides two orders of magnitudes higher bandwidths and is comparable to bandwidths of an Open MPI implementation.

Therefore we want to set the size limit for tiny messages as high as possible. The kernels run on a dual socket Intel Xeon 5675 with both processes pinned to one socket.

We activate our communication protocols when we transfer a new event. Since the number of outstanding message queue transfers is limited, we can encounter a *try again* error when we attempt to add a new message to the queue. As a result, we let communication strategies decide how to handle such a situation. Our default implementation retries sending until the receiver side drained enough events from the queue, such that the new message fits. For single threaded implementations the pushing is necessary as retaining the message could lead to deadlocks. Other implementations could use extra threads to reduce the performance impact of such a waiting situation. In summary we need to be able to buffer several messages in the message queue to do asynchronous communication and reach the queue size limit very rarely.

There are several ways to initialize the shared memory communication. The simple way is to use external specified knowledge about the partitioning of the used nodes, especially the number of processes within one shared memory domain. For MPI applications with equally distributed ranks the partitioning may be done based on rank number. One rank within a shared memory domain is used as tool rank, the other ranks are application ranks.

For many use cases there is no guarantee for a uniform distribution of ranks with equally-sized shared memory domains. We propose a protocol to achieve the detection of the shared memory domains and determine the tool process for the domains: Each process tries to create a message queue with a common key (e.g., an exported environment variable) that is unique for the invocation. The first process succeeds to create the message queue and becomes the leading tool process of the shared memory domain. The remaining processes detect that the queue already exists; thus they register with the tool process and open a communication channel with the tool process. The tool processes explore the distribution of the processes across the shared memory domains; they determine the total process count and verify that all processes registered to finish the registration stage. To build the TBON we need higher level tool processes, these are recruited from the smallest shared memory domains. Shared memory domains larger than a given limit get split into partitions with new first layer tool processes. At the end of the tree building step we assure that we have enough processes to provide a TBON with the required properties.

V. Buffering and Shutdown

If an application crash is handled, a runtime tool must:

- Ensure that all event buffering is aborted to assure that all events will be processed, and
- Provide a mechanism to shut down the tool and the application as gracefully as possible.

A. Buffering

Typical configurations of MUST—Figure 2(b)—rely on a communication strategy that aggregates multiple events into larger continuous buffers. This technique allows MUST to communicate bandwidth efficient and decreases its overhead in practice. Such communication strategies buffer events until either a larger buffer is full or the application issues its own shutdown. If an application failure occurs, such an application provided shutdown event is missing. Thus, communication strategies potentially buffer events indefinitely and thus violate Restriction C. An immediate solution is the usage of a communication strategy that does not use event buffering, but this solutions would increase tool overheads.

MUST layouts that support application crash handling still use buffering communication strategies—Figure 2(c)—between non-application layers. We then use additional events to turn off event buffering on all non-application nodes in case of an application failure:

- Crashed application processes inject a *panic* event and send it with their alternative communication medium towards the root,
- TBON nodes that receive a *panic* or *notifyPanic* event disable all event buffering and flush their strategies, and
- When the root receives the first *panic* event it broadcasts a *notifyPanic* event towards all non-application nodes in the TBON.

This handling efficiently notifies all non-application nodes that an application failure occured and disables event buffering as a result. A filter can remove redundant *panic* events on all layers of the TBON to avoid situations in which the root must receive and process high numbers of such events. Thus, TBON layers can use buffering communication. The only exception is the connection between the application processes and the first tool layer, since we avoid communication towards application processes (Assumption II). This layer must use an immediate communication stategy as a result, but this communication may still be asynchronous of course.

B. Shutdown

Assumption II from Section III challenges tool shutdown when failures occur. The tool must process all available events, but without communication to the application processes. Without such communication a tool can't guarantee that processes will not generate additional events. As a result, we propose a heuristic approach to handle tool shutdown. Figure 5 illustrates necessary communication within the TBON to vote for a shutdown: The beforementioned *panic* events initiate this alternative shutdown approach. Once first layer tool processes received an *notifyPanic* event, they start a timeout that they reset whenever they receive a new event. Once a tool process reaches its timeout it injects a *ack-shutdown* event that it sends towards the root of the TBON. This event contains a count of received, processed, and sent events. Each TBON layer adds up the numbers in these events and adds its own values. If the numbers match when the root receives the resulting sums, it terminates the tool with a *shutdown* event that it broadcasts throughout the TBON.

VI. Applicability

We use the Sierra cluster at the Lawrence Livermore National Laboratory to execute comparative measurements that evaluate the overheads of the purely asynchronous crash handling technique implemented in MUST. This system consists of 1,944 nodes where each node features two 6 core Xeon 5660 processors and 24 GB of main memory. The system uses a QDR InfiniBand interconnect.

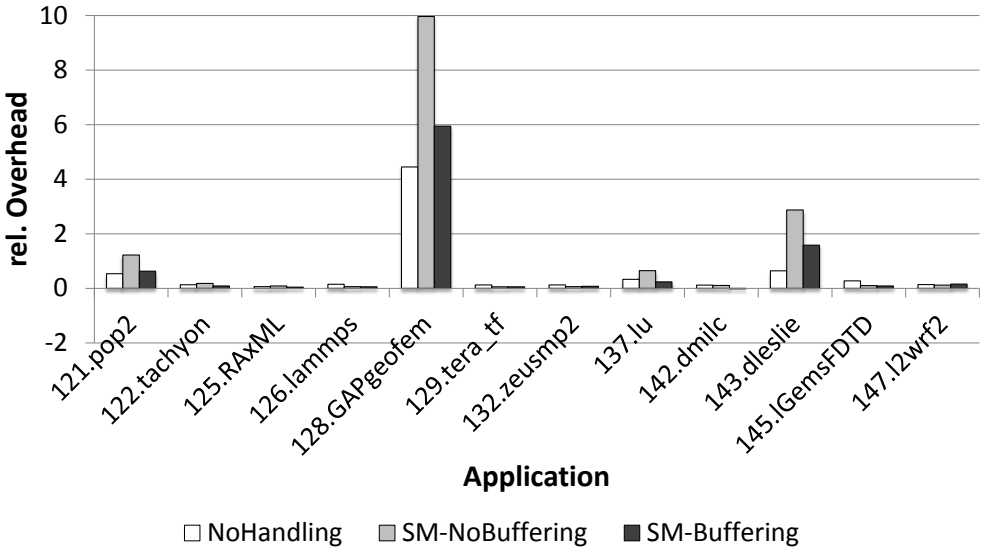


Figure 6. Slowdown comparison of a non crash tolerant version of MUST (*NoHandling*), a crash tolerant version of MUST without event buffering (*SM-NoBuffering*), and a crash tolerant version with buffering (*SM-Buffering*) for SPEC MPI2007 on Sierra.

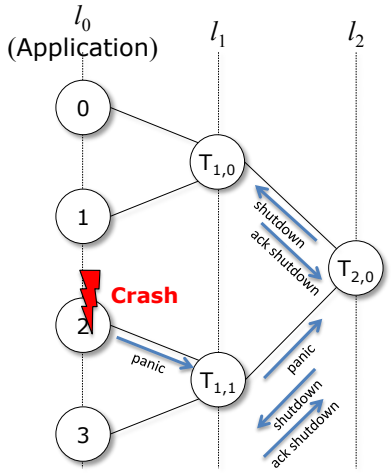


Figure 5. Shutdown protocol

We run the SPEC MPI2007 [12] (v2.0) benchmark with the *lref* data set to measure tool overheads of MUST for 256, 512, 1,024, and 2,048 processes. The MUST version [2] we use includes all of its correctness checks, except for its deadlock detection. We compare the following configurations:

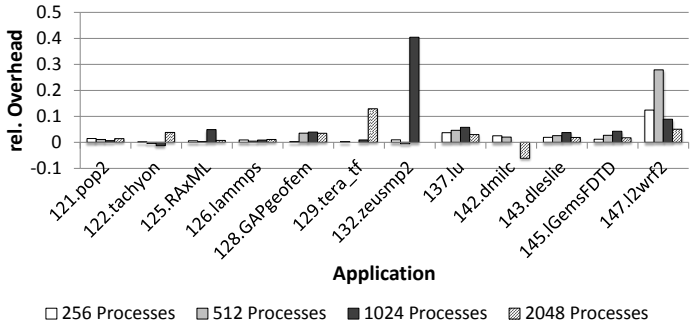
- Reference run with 12 processes per node (*ref12*);
- Reference run with 11 processes per node (*ref11*);
- Tool attached with no crash handling mechanism active (*NoHandling*);
- Tool attached with crash handling, but without buffering (*SM-NoBuffering*); and
- Tool attached with crash handling and with buffering (*SM-Buffering*).

We use the tool version *NoHandling* to compare runs with our crash handling technique to a well tested and optimized baseline that purely relies on MPI communication and buffering communication strategies (layouts as in Figure 2(b)). The tool version *SM-NoBuffering* uses shared memory communication, but only uses immediate communication strategies, i.e., it does not require the panic handling technique from Section V-A. The tool version *SM-Buffering* then uses shared memory communication and the panic handling scheme in order to employ aggregating communication strategies (layouts as in Figure 2(c)). As a result, a performance comparison between the latter two tool versions highlights the advantages of our panic handling.

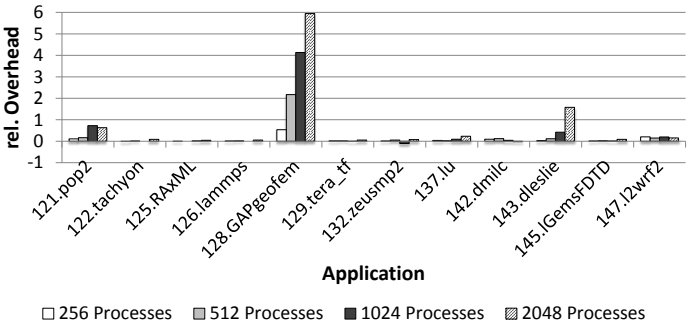
We use one run for each application scale, tool configuration, and application in a production system, i.e., with no chances for a compact node allocation and potential noise from neighboring jobs [13]. Thus, statistical significance of single measurements is limited, but should suffice for a performance tendency across all runs. The application *132.zeusmp* runs without failure for 1,024 and 2,048 processes, but internal application verification fails due to an effect that we want to study in the future. As a result, we use a wallclock runtime for these runs instead of the benchmark provided result times.

Figure 6 compares tool overheads with 2,048 processes for the different tool configurations. The charts use "relative overhead" as a metric that compares the runtime with a tool version to either the minimal runtime of the two reference runs; or to the runtime of another tool version. As an example, a value of 0.1 indicates that the application runs 10% longer with the specific version of the tool in comparison to either the reference runs or another tool version. The overall tendency shows the positive impact of event buffering. On the other hand the cost for crash handling is quite low.

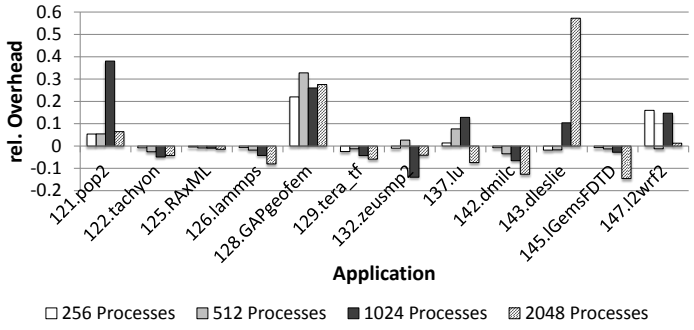
Without the presence of the tool, an application can use 12 MPI processes per compute node. For the shared memory communication system that we use for our crash handling, we place 11 application processes and one tool process onto each compute node. As a result, we use reference runs with 11 and 12 application processes and use the lower reference runtime in our runtime difference calculation for the tool versions. Figure 7(a) highlights that 11 processes per compute node does not increase runtimes for the applications, so the execution



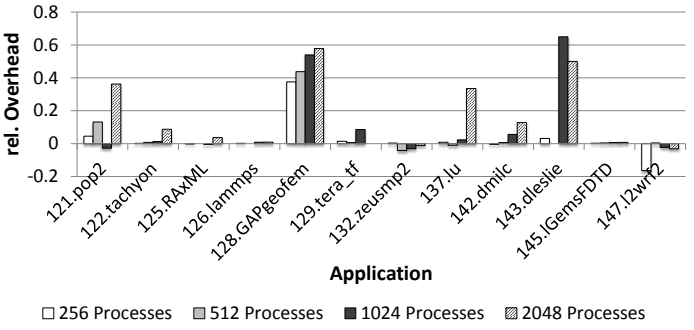
(a) Runtime increase for reference run when changing from 11 to 12 processes per node



(b) Overhead of crash-safe implementation compared to reference run



(c) Overhead of crash-safe implementation to tool without crash-handling.



(d) Overhead of non-buffering to buffering

Figure 7. Detailed comparison of overhead measurements

is not disturbed by this partitioning. In our measurements we observed runtime increases of up to 30% if we use 12 cores instead of 11 cores per compute node. Consequently we decided to base our overhead measurements on the minimal reference run of 11 and 12 processes per compute node. Besides the increases in application runtime, our tool configurations use additional compute cores as tool processes. Thus, our tool runs consume about 10% extra compute cores with the configurations that we use here.

Figure 7(b) highlights that all three tool versions exhibit noticeable overheads (more than 15% runtime increase) for the applications *128.GAPgeofem*, *121.pop2*, and *143.dleslie*. These applications issue MPI operations at a high rate that saturates the processing capabilities of our tool. Since SPEC MPI2007 is a strong scaling benchmark, the event rate increases with scale, and thus, the overhead of the tool versions. TBON layouts with additional tool processes—e.g., a tool process per 4 application processes—can reduce these overheads in practice.

Figure 7(c) details the overhead differences between the *NoHandling* tool version and the *SM-Buffering* version, i.e., the performance impact of our crash handling technique. The chart uses the runtime of the former version as the reference and details runtime increases/decreases for the tool version with crash handling. These values detail the cost of using shared memory instead of MPI communication as well as using an immediate rather than a buffering communication between the application layer and the first tool layer. We observe additional overhead with *SM-Buffering* for applications with high event rates. For the other cases, we observe a small decrease in runtime. As Figure 4 illustrates, the shared memory communication may compete with MPI communication if the receiver side promptly collects messages. When a tool process has a high load, collection delays can cause blocked message queues. The comparison in Figure 7(c) highlights a maximum runtime increase of 58% for the application *143.dleslie* with 2,048 processes. With 4% in average, the increase in application runtime for our tool version with crash handling (compared to our reference tool version) is almost neglectable.

Figure 7(d) illustrates that the *SM-NoBuffering* tool version yields up to 60% longer runtimes than the *SM-Buffering* version. Again, we see such overhead increases in cases which saturized the processing capabilities of the tool processes. This comparison demonstrates the huge impact of the panic handling scheme from Section V-A.

Finally, the benchmark applications of SPEC MPI2007 are well tested and cause no MPI related failures on Sierra. Thus, we used a test suite of incorrect synthetic MPI examples to evaluate whether our application crash handling scheme allows us to detect MPI usage errors that cause failures on application processes. Additionally, to test our approach at scale and for more complex applications, we inject MPI usage errors that cause application crashes for three of the SPEC MPI2007 applications at 2,048 processes:

- The use of an incorrect MPI communicator into the 100th `MPI_Isend` invocation of rank 1,025 for *121.pop2*,
- The use of an invalid communication buffer and count in the invocation of the 100th `MPI_Recv` of rank 777 for *137.lu*, and
- The use of an incorrect MPI reduction operation for the first `MPI_Allreduce` of rank 1 for *147.l2wrf2*.

The first two defects cause an application crash that triggers

a signal handler while the third triggers an MPI error handler. The tool version *SM-Buffering* successfully catches all three MPI usage errors and reports them in its error reports, even in the presence of an application failure. Especially the second and third type of defect stress our approach, since MUST can not locally detect these errors on the application processes, but only in its TBON hierarchy.

VII. Related work

Our work is both related and orthogonal to efforts for fault tolerant MPI implementations. Examples for fault tolerant MPI implementations are FT-MPI [14], FT/MPI [15], and P2P-MPI [16]. These approaches aim at failure situations where a computing unit gets disconnected from the application network, either by hardware fault or by network problems. The assumption is that the application execution is flawless. A fault tolerant MPI implementation typically aims to carefully remove the crashed rank (potentially losing information) from the application and then disables all communication to the crashed rank. FT/MPI implements a kind of checkpoint and restart mechanism for the crashed rank. P2P-MPI relies on redundant processing elements; the execution survives as long as not all replications of a rank died. FT-MPI leaves the task of data recovery to the application. Such mechanisms support our crash handling scheme with guarantees for Assumptions IIIa and IIIb. Where we currently rely on disjoint communication along with handlers that avoid a global application abort. At the same time, fault tolerant MPI implementations accept a loss of information on the application processes, whereas we must preserve all relevant events of crashed ranks. Thus, these approaches support an implementation of our approach, but they do not subsume it.

Marmot [10] is an MPI runtime checker written in C++ that covers MPI-1.2 and parts of MPI-2. The tool uses a single extra tool process as centralized "DebugServer" and implements the purely synchronous technique that we describe in Section III-A. For each intercepted MPI function call, Marmot performs two steps before executing the actual MPI call: first, it checks for correctness of the MPI call locally; second, it sends information on this MPI call to the *DebugServer*. The application process continues its execution only after it receives a ready-message from the *DebugServer*. As a result, it is guaranteed that all non-local checks executed at the *DebugServer*, as well as all local, are finished before the actual MPI call is issued. This synchronous checking ensures that all defects are reported before they can manifest in an application crash. In summary Marmot handles application failures in a platform and system indepenendent manner, but both suffers a scalability limitation from its centralized infrastructure as well as a performance penality from its communication and processing in the critical path.

The MPI correctness checker Umpire is written in C and has its focus on non-local MPI checks. It executes both a deadlock detection and type matching on a central manager. The first difference to Marmot is that Umpire spawns extra threads for each MPI process. Particularly, it spawns an "outfielder" thread as a communication thread on each process, as well as a "manager" thread on one process (usually process 0). The *manager* executes all non-local MPI checks and can be compared to Marmot's *DebugServer*. The *outfielder* thread asynchronously transfers event information to the *manager*. Application processes transfer MPI event information to their *outfielder* threads through process local shared memory, i.e., an alternative communication means. *Outfielder* threads aggregate event information and send it to the *manager* thread when a buffer limit is exceeded or when a timeout occurs. Depending on the system architecture, this communication is implemented with either MPI or System V IPC. Our approach in Section IV is inspired by this undocumented communication system. Umpire uses MPI for communication on distributed memory systems and requires an MPI implementation that supports the highest level of thread support [1] (`MPI_THREAD_MULTIPLE`). It uses MPI to communicate between the *outfielder* threads on all processes and the single *manager* thread. Umpire's communication system is designed to incur low runtime overhead, which is achieved with the asynchronous transfer of events to the central manager. Due to the asynchronous design, the central manager is no longer a bottleneck for communication. With respect to crash safety, Umpire registers signal handlers to fetch signals from operating system and MPI errors. When a failure occurs the application thread triggers the *outfielder* thread to flush buffered events. Umpire relies on the assumption that—at least outgoing—MPI communication is available for *outfielder* threads in case of a failure. Since these threads share the MPI implementation with an application process, this assumption may not hold if the implementation is in an inconsistent state due to a failure. In summary, Umpire's handling inspired our approach, but we do not require MPI implementations that support `MPI_THREAD_MULTIPLE`. Additionally, our assumption on independent MPI communication partitions (Assumption IIIb) is less restrictive then Umpire's assumption for communication on additional threads after a failure. The single *manager* thread of Umpire limits performance since it must analyze events from all processes. Further, performance tests with Umpire show that the efficiency of the asynchronous transfer depends highly on the interleaving of the communication of the application and the MPI communication of the *outfielder* threads [17].

The TBON infrastructure MRNet proposes *state compensation* [11] for failure recovery. The goal of the approach is that aggregating analyses within the TBON survive a TBON node failure without a loss of data. When a failure occurs, a recovery takes place that first, afterwards execution continues with a new TBON layout. The approach in MRNet specifically excludes failures on application processes and notes the need for failure handling on application processes. As a result, our approach complements the state compensation technique of MRNet and could provide both application and TBON node failure handling for MRNet or GTI based tools.

Debuggers such as Totalview [18] and DDT [19] typically fetch signals from the operating system and register breakpoints to routines that abort MPI applications, so the user is able to investigate error situations. The parallel debugger DDT uses a TBON infrastructure that builds on TCP as communication system and is independent from the applications MPI communication system. This TBON infrastructure is used to collect information to be presented on the graphical user interface, as well as to send commands to application processes. In summary, debuggers use similar techniques (error/signal handlers and alternative communication means) as in

our application crash handling. At the same time, commercial parallel debugging approaches with TBON infrastructures are proprietary and do not document their failure handling in detail. Additionally, the event rates that debuggers consider are far lower than our observation of all MPI communication operations, which requires an approach that provides high bandwidth communication.

VIII. Conclusions

We provided a categorization of approaches to handle application crashes in runtime analysis tools:

- A purely synchronous technique,
- A synchronous communication technique, and
- A completely asynchronous technique.

The latter only operates with techniques and use cases that satisfy several assumptions, but also provides lower overheads and better scalability. For an MPI tool with a TBON hierarchy, this requires an alternative communication system to decouple application communication from tool internal communication. Additionally, we propose a shutdown protocol to integrate buffering to the tool internal communication and verified the performance impact of this technique. Benchmark measurements of our implementation of the completely asynchronous technique highlight low overheads in most cases resulting in an avarage runtime increase of 4% for our MPI correctness tool. Moderate runtime increases of up to 60% occur for applications with very high event rates and identify a potential for future optimizations in our alternative communication means implementation. We demonstrated the applicability of our approach with up to 2,048 application processes; our implementation was able to detect MPI usage errors for three different injected failures at this scale.

Acknowledgments

We thank the ASC Tri-Labs and the Los Alamos National Laboratory for their friendly support. Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-??????). This work has been supported by the CRESTA project that has received funding from the European Community's Seventh Framework Programme (ICT-2011.9.13) under Grant Agreement no. 287703.

References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 2.2," http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf, 2012, last visited on 28/04/2014.

[2] T. Hilbrich, B. R. de Supinski, W. E. Nagel, J. Protze, C. Baier, and M. S. Müller, "Distributed Wait State Tracking for Runtime MPI Deadlock Detection," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'13. New York, NY, USA: ACM, 2013, pp. 16:1–16:12.

[3] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003.

[4] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: A Tool for Model Checking MPI Programs," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07. New York, NY, USA: ACM, 2008, pp. 285–286.

[5] D. C. Arnold, D. H. Ahn, B. R. de Supinski, G. L. Lee, B. P. Miller, and M. Schulz, "Stack Trace Analysis for Large Scale Debugging," *International Parallel and Distributed Processing Symposium*, vol. 0, 2007.

[6] M. Gerndt, K. Fürlinger, and E. Kereku, "Periscope: Advanced Techniques for Performance Analysis," in *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[7] T. Hilbrich, M. S. Müller, B. R. de Supinski, M. Schulz, and W. E. Nagel, "GTI: A Generic Tools Infrastructure for Event-Based Tools in Parallel Systems," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1364–1375.

[8] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs," in *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., ZIH. Springer Publishing Company, Incorporated, 2009.

[9] M. Schulz, D. H. Ahn, A. Bernat, B. R. de Supinski, S. Y. Ko, G. L. Lee, and B. Rountree, "Scalable dynamic binary instrumentation for Blue Gene/L," *SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 9–14, 2005.

[10] B. Krammer and M. S. Müller, "MPI Application Development with MARMOT," in *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, ser. John von Neumann Institute for Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, 2005, pp. 893–900.

[11] D. C. Arnold and B. P. Miller, "Scalable Failure Recovery for High-Performance Data Aggregation," in *Proceedings of the 2010 IEEE 24th International Parallel and Distributed Processing Symposium*, ser. IPDPS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[12] M. S. Müller, M. van Waveren, R. Lieberman, B. Whitney, H. Saito, K. Kumaran, J. Baron, W. C. Brantley, C. Parrott, T. Elken, H. Feng, and C. Ponder, "SPEC MPI2007 – An Application Benchmark Suite for Parallel Systems using MPI," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 2, pp. 191–205, 2010.

[13] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 41:1–41:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503247

[14] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An Evaluation of User-Level Failure Mitigation Support in MPI," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 193–203. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33518-1_24

[15] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware," *Cluster Computing*, vol. 7, no. 4, pp. 303–315, 2004.

[16] S. Genaud, E. Jeannot, and C. Rattanapoka, "Fault-Management in P2P-MPI," *International Journal of Parallel Programming*, vol. 37, no. 5, pp. 433–461, 2009.

[17] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller, "A Graph Based Approach for MPI Deadlock Detection," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 296–305.

[18] Rogue Wave Software, "Totalview Graphical Debugger," http://www.roguewave.com/products/totalview.aspx, last visited on 28/04/2014.

[19] Allinea Software, "Allinea DDT," http://www.allinea.com/products/ddt/, last visited on 28/04/2014.